Computer Security and Privacy Course Project Spring 2023

Rohit Vasishta and Gautam Yajaman

Contents

1 Abstract

The Discrete Algorithm is an analogue of the ordinary logarithm on in a finite Abelian group. Suppose $(\mathcal{H}, \otimes)$ is a finite Abelian group, and $\mathcal{G}$ is a cyclic subgroup of $\mathcal{H}$ generated by element $g \in mathcalH$. In other words, we have $< g >= \mathcal{G}$. Given $a, b \in \mathcal{G}$, the The Discrete Logarithm problem is to find the smallest non-negative integer α such that

$$h = g \otimes g \otimes g \otimes \cdots \otimes g = g^\alpha$$

Since $\mathcal{G}$ is a cyclic group subgroup of a finite group, it will have a finite order, and therefore, such an α surely exists and is unique. It is a hard problem in certain groups, and easy to compute in others. In this course project, we aim to

1. Understand and help the reader understand why the DLP is hard to compute.
2. Implement 4 known algorithms for computing the discrete logarithm.
3. Compare each algorithm with respect to time and space complexity.
4. Compare each algorithm on the type of group structure and other pre-requisites required for its implementation.
5. Present the lower bound time complexity of the discrete logarithm problem.

2 Algorithms for computing Discrete Logarithm

2.1 Discrete Logarithms in Additive Groups $(\mathbb{Z}_n^{,}+)$

2.1.1 Algorithm Description

Suppose we have $GCD(\alpha, n) = 1$. This implies that α is a generator of $\mathbb{Z}_n$. The Discrete Logarithm a of β in this group corresponds to finding a such that

$$\alpha a \equiv \beta (\mod n)$$
$$\implies a = \log_\alpha \beta \equiv \beta\alpha^{-1} \mod n$$

One can compute such a value of a easily using Euclid's Extended Algorithm. This can be done in $O(\log n)$ time.

2.1.2 A Theorem Establishing Isomorphism between Cyclic Subgroups of same order

Two cyclic groups of the same finite order are isomorphic to each other.

Proof: Let G_1, G_2 by cyclic groups with order $k < \infty$. Let

$$G_1 =< a > \text{ and } G_2 =< b >$$

such that $|a| = |b| = k$. Clearly, we can write

$$G_1 = \{a^0, a^1, ..., a^{k-1}\} \text{ and } G_2 = \{b^0, b^1, ...b^{k-1}\}$$

Then, the map

$$\Phi : G_1 \to G_2 : \Phi(a^i) = b^i \qquad \forall 0 \le i \le k-1$$

is an isomorphism between the two groups. We show this in the following manner: For any $x, y \in G_1$, we know that

$$G_1 =< a > \implies \exists s, t \in \mathbb{Z} \ni x = a^s, y = a^t$$

and we have

$$\begin{aligned}\Phi(xy) &= \Phi(a^s a^t)\\ &= \Phi(a^{s+t})\\ &= b^{s+t}\\ &= \Phi(a^s)\Phi(a^t)\\ &= \Phi(x)\Phi(y)\end{aligned}$$

So, Φ is a Homomorphism. Further Φ is clearly bijective. Therefore, Φ is an isomorphism and hence, G_1, G_2 are isomorphic. □

2.1.3 Dual of the Discrete Logarithm Problem - motivating the hardness

In the first sub-section, we found an efficient way to compute the discrete logarithm of a cyclic additive group. Using the theorem presented above, we know that there will exist an isomorphism between the additive group and any other cyclic subgroup with the same order. Therefore, finding the discrete logarithm in these sub-groups can be made extremely efficient if we just found an isomorphism between them. If this were possible, discrete logarithms can be computed extremely fast through the Extended Euclidean Algorithm and a few more mathematical machinery. Consider the following scenario:

We are given a subgroup $< \alpha >$ of order n of G. We are required to find the discrete logarithm $x = \log_\alpha \beta$ of any element $\beta \in < \alpha >$.
Suppose $< \alpha >$ is isomorphic to $(\mathbb{Z}_n, +)$ through the isomorphic map Φ. Then, we have

$$\Phi(xy) = (\Phi(x) + \Phi(y)) \mod n$$

This implies that for any $x, y \in < \alpha >$, we have

$$\begin{aligned} & \Phi(\alpha^a) = a\Phi(\alpha) \mod n\\ \beta = \alpha^a \iff & a\Phi(\alpha) \equiv \Phi(\beta)(\mod n)\\ \implies & \log_\alpha \beta = \Phi(\beta)\left(\Phi\alpha\right)^{-1} \mod n\end{aligned}$$

We can easily compute the last discrete logarithm through the extended Euclidean Algorithm. It is only a matter of finding a valid Isomorphism between the two groups. As fate would have it, finding such an isomorphism is unfortunately, extremely hard. Therefore, we turn to the following algorithms to find the discrete logarithm in any multiplicative cyclic group.

2.2 Baby Step Giant Step

Proposed by the American Mathematician Daniel Shanks in his paper ”Class number, a theory of factorization and genera”, this is a general algorithm for every finite cyclic group that works naively in finding the discrete logarithm by hit and trial multiplication, combined with a table-lookup.

2.2.1 Pre-requisites

1. Group structure: Every finite cyclic group
2. Group order is not required
3. Preferably, prime order group (though variations of this algorithm exist for non-prime order groups)

2.2.2 Algorithm: Pseudocode

Algorithm 1 Baby-Step-Giant-Step/Shanks

```
Require: Group $G$ of order $n$, generator $\alpha$ and element $\beta$. Output: $x \ni \alpha^x = \beta$
Ensure: $x = im + j$, where $m = \lceil \sqrt{n} \rceil \forall 0 \leq i, j \leq m$.
  $m \leftarrow \lceil \sqrt{n} \rceil$
  for $0 \leq j \leq m$ do:
      $y \leftarrow \alpha^j$
      Store $(j, y)$ in table T
  $z \leftarrow \alpha - m$
  $\gamma \leftarrow \beta$
  for $0 \leq i < m$ do
      if $\alpha^j \in T$ then
          return $im + j$
          Else $\gamma \leftarrow \gamma \cdot \alpha^{-m}$
```

A slight modification of the algorithm above, given by Shanks provides a more efficient computation of the discrete logarithm.

2.2.3 Modification of Baby-Step Giant Step: Shanks' Algorithm

```
1: function SHANKS($G, n, \alpha, \beta$)
2:     $m \leftarrow \lfloor \sqrt{n} \rfloor$
3:     for $j = 0$ to $m - 1$ do
4:         Compute $\alpha^{mj}$
5:     Sort $m$ ordered pairs $(j, \alpha^{mj})$ with respect to the second coordinate, obtaining list $L_1$
6:     for $i = 0$ to $m - 1$ do
7:         Compute $\beta\alpha^{-i}$
8:     Sort the $m$ ordered pairs $(i, \beta\alpha^{-i})$ with respect to the second coordinates, obtaining $L_2$
9:     Find pair $(j, y) \in L_1$ and $(i, y) \in L_2$
10:    return $(mj + i) \bmod n$
```

2.2.4 Proof of Correctness

Claim: We claim that the algorithm returns the correct value of the discrete logarithm, i.e given $\beta \in \langle \alpha \rangle$, the algorithm returns a unique exponent $0 \leq a \leq n - 1$ such that $\alpha^a = \beta$. Note that via the algorithm, we have found

$$(j, y) \in L_1, (i, y) \in L_2$$

This implies

$$\begin{aligned} \alpha^{mj} &= y = \beta\alpha^{-i} \\ \implies \alpha^{mj}\alpha^{i} &= \beta\alpha^{-i}\alpha^{i} \\ \implies \alpha^{mj+i} &= \beta \end{aligned}$$

Conversely, for any $\beta \in \langle\alpha\rangle$, we have $0 \leq \log_{\alpha}^{\beta} \leq n-1$. This is because:

$$\begin{aligned} \lfloor m^{-1}\log_{\alpha}^{\beta}\rfloor &= j \\ \implies \log_{\alpha}^{\beta} &= mj + i \forall 0 \leq i, j \leq m-1 \end{aligned}$$

2.2.5 Complexity and further notes

Shanks' algorithm cannot perform in an order below $O(m)$. This is because steps $2, 6, 9$ necessitate the the action of traversing a list of size m. The order of the algorithm therefore depends on the time complexity of steps $5, 8$, which is a problem of optimizing list sorting. Using Merge Sort as the de-facto sorting algorithm leads to an algorithmic running time order of $O(m \log m)$.

2.2.6 Implementation

```python
import math
import helpers

def shanks(n,alpha,beta):
    result = None
    j_table = []
    m = math.ceil(math.sqrt(n))
    for j in range(m):
        temp = (alpha**(m*j)) % n
        j_table.append((j, temp))
    inverse_alpha = helpers.modular_inverse(alpha, n)

    j_table = helpers.merge_sort(j_table)

    temp = beta
    i_table = []
    for i in range(m):
        i_table.append((i,temp))
        temp = (temp*inverse_alpha) % n

    i_table = helpers.merge_sort(i_table)

    for item in j_table:
        for element in i_table:
            if item[1] == element[1]:
                result = (m*item[0] + element[0]) % n
                return result
```

2.3 Pollard's Rho Algorithm for Discrete Logarithms

Pollard's Rho Algorithm for Discrete Logarithms is randomized algorithm for computing discrete logarithms. It works by defining a pseudo-random sequence and then detecting a match in the sequence. The match will lead to a solution of the discrete logarithm problem with high probability. The two key ideas involved are the iteration function for generating the sequence and a cycle-finding algorithm for detecting a match.

2.3.1 Pre-requisites

In his original paper, Pollard considered DLP's in the groups $\mathbb{Z}/p\mathbb{Z}$ where p is prime.

2.3.2 Algorithm: Pseudocode

Algorithm 2 Pollard's Rho Algorithm for Discrete Logarithms(G, n, α, β)

```
1: procedure F$(x, a, b)$
2:     if $x \in S_1$ then
3:         $f \leftarrow (\beta x, a, (b+1) \bmod n)$
4:     else if $x \in S_2$ then
5:         $f \leftarrow (x^2, 2a \bmod n, 2b \bmod n)$
6:     else
7:         $f \leftarrow (\alpha x, (a+1) \bmod n, b)$
8:     return $f$

9: procedure Computing the Discrete Logarithm
10:     Define the partition $G = S_1 \cup S_2 \cup S_3$
11:     $(x, a, b) \leftarrow f(1, 0, 0)$
12:     $(x_0, a_0, b_0) \leftarrow f(x, a, b)$
13:     while $x \neq x_0$ do
14:         $(x, a, b) \leftarrow f(x, a, b)$
15:         $(x', a', b') \leftarrow f(x', a', b$
16:         $(x', a', b') \leftarrow f(x', a', b')$
17:         if $\gcd(b_0 - b, n) \neq 1$ then
18:             return "failure"
19:         else
20:             return $((a - a_0)(b_0 - b)^{-1} \bmod n)$
```

2.3.3 Proof of Correctness

Pollard's Rho Algorithm works on the guarantee of finding a match in the sequence. Since we are computing an infinite sequence of values, all of which are members of a finite group, we are bound by the Pigeonhole principle to land on a match.

2.3.4 Complexity and further notes

In $\mathbb{Z}/p\mathbb{Z}$ groups, Pollard's Rho Algorithm has an average time complexity of $O(\sqrt{n})$, where n is the order of the group. This assumes that the function f is considerably random.

2.3.5 Implementation

```python
import helpers

def pollard_rho(n,alpha,beta):

    def f(x):
        if x % 3 == 0:
            return (x*x) % n
        if x % 3 == 1:
            return (alpha*x) % n
        if x % 3 == 2:
            return (beta*x) % n

    def g(x,k):
        if x % 3 == 0:
            return (2*k) % (n-1)
        if x % 3 == 1:
            return (k + 1) % (n-1)
        if x % 3 == 2:
            return k

    def h(x,k):
        if x % 3 == 0:
            return (2*k) % (n-1)
        if x % 3 == 1:
            return k
        if x % 3 == 2:
            return (k + 1) % (n-1)

    a = [None for i in range(n*2)]
    b = [None for i in range(n*2)]
    x = [None for i in range(n*2)]

    a[0] = 0
    b[0] = 0
    x[0] = 1

    i = 1

    while(True):
        x[i] = f(x[i-1])
        a[i] = g(x[i-1], a[i-1])
        b[i] = h(x[i-1], b[i-1])

        x[2*i] = f(f(x[2*i-2]))
        a[2*i] = g(f(x[2*i-2]), g(x[2*i-2], a[2*i-2]))
        b[2*i] = h(f(x[2*i-2]), h(x[2*i-2], b[2*i-2]))

        if x[i] == x[2*i]:
            r = b[i] - b[2*i]
            if r == 0:
                return None
            result = ((helpers.modular_inverse(r,n))*(a[2*i] - a[i])) % (n)
            return result
        else:
            i = i+1
```

2.4 Pohlig-Hellman Algorithm

Before we state and explain the Pohlig-Hellman Algorithm, we wish to remind/introduce the reader to the **Chinese Remainder Theorem:**

2.4.1 The Chinese Remainder Theorem

Let $m_1, m_2, \ldots, m_k$ be pairwise co-prime positive integers and let $a_1, a_2, \ldots, a_k$ be any integers. Then the system of congruences

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_k \pmod{m_k} \end{aligned}$$

has a unique solution modulo $m_1 m_2 \cdots m_k$.

2.4.2 Pre-requisites

- A prime p such that $p-1$ has small prime factors.
- An element α of order n in $\mathbb{Z}_p^*$.
- An element β in $\mathbb{Z}_p^*$.
- A factorization of n into primes $n = q_1^{c_1} \cdot q_2^{c_2} \cdot \cdots \cdot q_k^{c_k}$.

2.4.3 Algorithm: Pseudocode

Here is the pseudocode for the algorithm.

Algorithm 3 Pohlig-Hellman

```
1:  procedure POHLIG-HELLMAN($(G, n, \alpha, \beta, q, c)$)
2:      $j \leftarrow 0$
3:      $\beta_j \leftarrow \beta$
4:      while $j \leq c-1$ do
5:          $\delta \leftarrow \beta_j^{n/q}$
6:          find $i$ such that $\delta = \alpha^{aj}$
7:          $a_j \leftarrow i$
8:          $\beta_{j+1} \leftarrow \beta_j \alpha^{-ajq}$
9:          $j \leftarrow j+1$
10:     return $(a_0, \ldots, a_{c-1})$
```

2.4.4 Algorithm: Explanation and Proofs

By the Fundamental Theorem of Arithmetic, any number n can be factored into a product of primes $p_1, p_2, ..., p_k$. In other words, we can express

$$n = \prod_{i=1}^{k} p_i^{c_i}$$

The discrete logarithm $a = \log_\alpha^\beta$ can be computed via the Pohlig-Hellman algorithm. First, we notice that if one can compute $a \mod p_i^{c_i}$ for all $1 \leq i \leq k$, we can compute $a \mod n$ by the Chinese Remainder Theorem stated above. Now, suppose q is prime. We need to choose a group such that

$$n \equiv 0(\mod q^c) \text{ and } n \not\equiv 0(\mod q^{c+1})$$

For $0 \leq x \leq q^c - 1$, we can compute the value $x = a \mod q^c$.The algorithm exploits the radix-q representation of x, which is

$$x = \sum_{i=0}^{c-1} a_i q^i$$

where

$$0 \leq a_i \leq q - 1 \text{ and } 0 \leq i \leq c - 1$$

Therefore, we can write

$$a = x + sq^c; s \in \mathbb{Z}$$

So we can write

$$a = \sum_{i=0}^{c-1} a_i q^i + sq^c$$

Now, if $a = \log_\alpha^\beta$, we will have $\beta = \alpha^a$. We find

$$\begin{aligned} \beta^{\frac{n}{q}} &= (\alpha^a)\frac{n}{q} \\ &= \alpha^{\frac{(a_0+a_1q+\cdots+a_{c-1}q^{c-1}+sq^c)n}{q}} \\ &= \alpha^{a_0+Kq}\frac{n}{q} \\ &= \alpha^{a_0\frac{n}{q}}\alpha^{Kn} \\ &= \alpha^{a_0\frac{n}{q}} \end{aligned}$$

Our intention is to find a_0. Let $\gamma = \alpha^{n/q}$ Now, notice that for some $i \leq q - 1$, we will have

$$\gamma^i = \beta^{\frac{n}{q}}$$

then $a_0 = i$. Now, if $c = 1$,by step 4 of the pseudocode, the algorithm will terminate. Else, we will find $a_1, a_2, ..., a_{c-1}$ through the following recurrence relations (3) and (4) listed below:

$$\beta_0 = 0 \tag{1}$$

$$\beta_j = \beta\alpha^{-(a_0+a_1q+...+a_{j-1}q^{j-1})} \tag{2}$$

$$\beta_j^{\frac{n}{q^{j+1}}} = \alpha^{a_j\frac{n}{q}} \tag{3}$$

$$\beta_{j+1} = \beta_j\alpha^{-a_jq^j} \tag{4}$$

Once done, we return the discrete logarithm using the radix-q representation as follows:

$$a = x + sq^c = \sum_{i=0}^{c-1} a_i q_i + sq^c$$

Clearly, this algorithm terminates with the correct value of the discrete logarithm.

2.4.5 Complexity and Further Notes

The implementation of the Pohlig-Hellman algorithm according to the pseudocode given above will terminate with the value of the discrete logarithm in $O(cq)$. A creative approach toward making this more efficient arises out of viewing δ as a solution of a particular instance of the Discrete Logarithm Problem, i.e

$$\delta = \alpha^{in/q} \iff i = \log_{\alpha^{n/q}} \delta$$

Now, observe that since $\alpha^{n/q}$ has order q, each i can be calculated using Shanks' algorithm in $O(\sqrt{q})$ time. Therefore, the optimized version of the Pohlig-Hellman algorithm, when combined with Shanks gives rise to a time complexity of $O(c\sqrt{q})$

2.4.6 Implementation

```python
from babystepgiantstep import babystepgiantstep
import helpers
import math

def prime_factors(n):
    factors = []
    while n % 2 == 0:
        factors.append(2)
        n //= 2

    for i in range(3, int(math.sqrt(n)) + 1, 2):
        while n % i == 0:
            factors.append(i)
            n //= i

    if n > 2:
        factors.append(n)
    return factors

def factor_powers(n):
    factors = prime_factors(n)
    result = {}
    for factor in factors:
        if factor in result:
            result[factor] += 1
        else:
            result[factor] = 1
    return result

def chinese_remainder_theorem(moduli, remainders):
    product = 1
    for modulus in moduli:
        product *= modulus
    N = product
    x = 0
    for mi, ri in zip(moduli, remainders):
        Ni = N // mi
        xi = helpers.modular_inverse(Ni, mi)
        x += ri * xi * Ni
    return x % N

def pohlig_hellman(n, alpha, beta):
    factorization = factor_powers(n)
    moduli = []
    remainders = []
    for p, e in factorization.items():
        pe = p ** e
        n_i = n // pe
        alpha_i = pow(alpha, n_i, n)
        beta_i = pow(beta, n_i, n)
        x_i = babystepgiantstep(pe, alpha_i, beta_i)
        moduli.append(pe)
        remainders.append(x_i)
    return chinese_remainder_theorem(moduli, remainders)
```

2.5 Index Calculus Algorithm

2.5.1 Pre-requisites

The Index Calculus is an example of a non-generic algorithm to compute the discrete logarithm. In other words, this algorithm cannot be applied onto all groups. Rather, it only applies to the situation of finding discrete logarithms in $\mathbb{Z}_p^*$, where p is a prime number and α is the primitive element modulo p of $\mathbb{Z}_p^*$. The algorithm proceeds in two phases- a pre-computation phase and the computational phase. We describe both phases with a running example below:

2.5.2 Pre-computation Phase: Algorithm Steps and Running Example

1. Choose a prime p, an element β and a generator α.
2. The algorithm requires a factor base

$$\mathcal{B} = \{p_1, p_2, ..., p_B\}$$

   a set of B prime numbers.
3. Now, pick a number $C > B$. Pick a random value x. Compute $\alpha^x \mod p$ and determine if $\alpha^x \mod p$ has all its factors in $\mathcal{B}$. Now, observe that this can be equivalently written in the

following form- we need to construct C congruences modulo p such that

$$\alpha^{x_j} \equiv p_1^{a_{1j}} p_2^{a_{2j}} \cdots p_B^{a_{Bj}} \qquad \forall 1 \leq j \leq c$$

$$\implies x_j \equiv a_{1j} \log_\alpha p_1 + a_{2j} \log_\alpha p_2 + \cdots a_{Bj} \log_\alpha p_b (\mod p-1) \qquad \forall 1 \leq j \leq c$$

$$\implies \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_C \end{bmatrix} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{B1} \\ a_{12} & a_{22} & \cdots & a_{B2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1C} & a_{2C} & \cdots & a_{BC} \end{bmatrix} \begin{bmatrix} \log_\alpha p_1 \\ \log_\alpha p_1 \\ \vdots \\ \log_\alpha p_B \end{bmatrix} \quad \text{mod } p-1$$

4. If there exists a unique solution to the above system of equations, we can compute the logarithms of the elements in the factor base. The pre-computation step ends here.

Now, we present a running example of the pre-computation step:

1. We pick $p = 10007$ and $\alpha = 5$
2. Suppose $\mathcal{B} = \{2, 3, 5, 7\}$ is the factor base
3. Pick $x = \{4063, 5136, 9865\}$. We must compute $5^x \mod 10007$. With each of the $x's$ we have picked, we find

$$5^{4063} \mod 10007 = 42 = 2(3)(7) \tag{5}$$
$$5^{5136} \mod 10007 = 54 = 2(3)^3 \tag{6}$$
$$5^{9865} \mod 10007 = 189 = (3)^3(7) \tag{7}$$

using which we derive the system of equations

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 3 & 0 \\ 3 & 1 & 0 \end{bmatrix} \begin{bmatrix} \log_5 2 \\ \log_5 3 \\ \log_5 7 \end{bmatrix} = \begin{bmatrix} 4063 \\ 5136 \\ 9865 \end{bmatrix} \quad \text{mod } 10006 \tag{8}$$

The matrix

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 3 & 0 \\ 3 & 1 & 0 \end{bmatrix}$$

has a non-zero determinant, implying that it is of full-rank, and hence, it is clear that the system of equations (8) will have a unique solution.

4. It so turns out that we find the unique solution to be

$$\begin{bmatrix} \log_5 2 \\ \log_5 3 \\ \log_5 7 \end{bmatrix} = \begin{bmatrix} 6578 \\ 6190 \\ 1301 \end{bmatrix}$$

The pre-computation step is done.

2.5.3 Computation Phase: Algorithm and Running Example

Having calculated the logarithms of all the numbers in the factor base $\mathcal{B}$ to the base α, we now wish to compute the discrete logarithm $\log_\alpha \beta$. Now, the algorithm proceeds as follows:

1. Use a randomnized algorithm that always returns correct output, and might compromise on running time to choose a random integer $s \in \mathbb{Z}$ such that $1 \leq s \leq p-2$.
2. Let $\gamma = \beta\alpha^s \mod p$. Assuming that *gamma* can be factored using the Fundamental Theorem of Arithmetic using the factors present in the factor base $\mathcal{B}$, we find

$$\beta\alpha^s \equiv p_1^{c_1} p_2^{c_2} \cdots p_B c^B \mod p \tag{9}$$

$$\implies \log_\alpha \beta \equiv \sum_{i=1}^{B} c_i \log_\alpha p_i - s(\mod p-1) \tag{10}$$

Since we know the values of $\log_\alpha p_1, \ldots \log_\alpha p_B$, it is just a matter of arithmetic computation to return the discrete logarithm $\log_\alpha \beta$! We continue the running example presented in the previous section:

1. Suppose we choose $s = 7736$
2. Suppose $\beta = 9451, \alpha = 5$. We wish to compute $\log_\alpha \beta$. We compute

$$\gamma = 9451 \times 5^{7736} \mod 10007 = 8400 = 2^4 3^1 5^2 7^1$$

Therefore, we find

$$\begin{aligned}\log_5 9451 &= 4\log_5 2 + \log_5 3 + 2\log_5 5 + 4\log_5 7 - s \mod 10006 \\ &= 6057\end{aligned}$$

Therefore, the algorithm has terminated, returning the value $\log_5 9451 = 6057$.

2.5.4 Complexity and Further Notes

It has been found that the following are the running time complexities of the algorithm's phases:

1. Pre-computation phase: $O\left(e^{((1+o(1))\sqrt{\ln p \ln \ln p}}\right)$
2. Computation phase: $O\left(e^{(1/2+o(1))\sqrt{\ln p \ln \ln p}}\right)$

2.5.5 Implementation

```python
import random
from sympy import primefactors
import helpers
import math

def generate_relations(p, g, factor_base, num_relations):
    relations = []
    x_values = []
    while len(relations) < num_relations:
        x = random.randint(1, p - 1)
        gx = helpers.mod_exp(g, x, p)
        factors = primefactors(gx)
        if all(f in factor_base for f in factors):
            relations.append(factors)
            x_values.append(x)
    return relations, x_values

def build_matrix(p, factor_base, relations):
    matrix = []

    for relation in relations:
        row = [0] * len(factor_base)
        for f in relation:
            row[factor_base.index(f)] = (row[factor_base.index(f)] + 1) % 2
        matrix.append(row)

    return matrix

def gaussian_elimination(matrix):
    row, col = 0, 0
    while row < len(matrix) and col < len(matrix[0]):
        if matrix[row][col] == 0:
            for r in range(row + 1, len(matrix)):
                if matrix[r][col] == 1:
                    matrix[row], matrix[r] = matrix[r], matrix[row]
                    break
            else:
                col += 1
                continue

        for r in range(row + 1, len(matrix)):
            if matrix[r][col] == 1:
                matrix[r] = [(matrix[r][c] - matrix[row][c]) % 2 for c in range(len(matrix[0]))]

        row += 1
        col += 1
    return matrix

def find_log(p, g, h, factor_base, matrix, x_values):
    for row in matrix:
        if sum(row) == 1:
            i = row.index(1)
            x = x_values[matrix.index(row)]
            log_g_h = x * factor_base[i] % (p - 1)
            if helpers.mod_exp(g, log_g_h, p) == h:
                return log_g_h

def index_calculus(p, g, h, bound, num_relations):
    factor_base = [-1] + primefactors(p-1)[:bound]
    relations, x_values = generate_relations(p, g, factor_base, num_relations)
    matrix = build_matrix(p, factor_base, relations)
    reduced_matrix = gaussian_elimination(matrix)
    result = find_log(p, g, h, factor_base, reduced_matrix, x_values)
    return result
```

2.6 Helper Functions

Below is a list of helper functions used in the implementation of the above algorithms. These are relatively simple functions that are already well documented.

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    left = merge_sort(left)
    right = merge_sort(right)

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i][1] < right[j][1]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result

def modular_inverse(a, n):
    r0, r1 = a, n
    x0, x1 = 1, 0
    y0, y1 = 0, 1

    while r1 != 0:
        q = r0 // r1
        r0, r1 = r1, r0 - q * r1
        x0, x1 = x1, x0 - q * x1
        y0, y1 = y1, y0 - q * y1

    if r0 != 1:
        return None
    else:
        return x0 % n

def mod_exp(base, exp, mod):
    result = 1
    while exp > 0:
        if exp % 2 == 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp //= 2
    return result
```

2.7 Comparative Analysis of Algorithms: Observations

2.7.1 Baby Step Giant Step

```
PS C:\Users\gausi\OneDrive\Documents\School\Spring 2023\Computer Security and Privacy
\Final Project> python3 babystepgiantstep.py
p: 1419857  g: 5  h: 41
Computed x: 737135
Successfully found the discrete logarithm!
Time elapsed = 0.043892860412597656
```

2.7.2 Shanks' Algorithm

Although our implementation of Shanks' algorithm does not deviate from the specifications, it is much slower than our implementation of Baby Step Giant Step.

```
PS C:\Users\gausi\OneDrive\Documents\School\Spring 2023\Computer Security and Privacy
\Final Project> python3 shanks.py
p: 1419857  g: 5  h: 41
Computed x: 737135
Successfully found the discrete logarithm!
Time elapsed = 10.529232025146484
```

2.7.3 Pollard's Rho Algorithm

Although our implementation of Pollard's Rho algorithm does not deviate from the specifications, it unfortunately does not appear to give the correct result for every single input. This means that comparing its running time to our other algorithms is not possible. Below is an example of it's running time for an input that it works for.

```
PS C:\Users\gausi\OneDrive\Documents\School\Spring 2023\Computer Security and Privacy
\Final Project> python3 pollard_rho.py
10
p: 1019  g: 2  h: 5
Computed x: 1019
Successfully found the discrete logarithm!
Time elapsed = 0.0
```

2.7.4 Pohlig-Hellman

```
PS C:\Users\gausi\OneDrive\Documents\School\Spring 2023\Computer Security and Privacy
\Final Project> python3 pohlig_hellman.py
p: 1419857  g: 5  h: 41
Computed x: 737135
Successfully found the discrete logarithm!
Time elapsed = 0.031250715255737305
```

2.7.5 Index Calculus

```
PS C:\Users\gausi\OneDrive\Documents\School\Spring 2023\Computer Security and Privacy
\Final Project> python3 index_calculus.py
FAIL
FAIL
p: 49  g: 5  h: 41
Computed x: 15
Successfully found the discrete logarithm!
Time elapsed = 0.015650033950805664
```

Since Index Calculus is a probabilistic algorithm, it may not always return a result. If we use the same input as the previous algorithms, we see that the computation of the discrete log takes a very long time.

```
PS C:\Users\gausi\OneDrive\Documents\School\Spring 2023\Computer Security and Privacy
\Final Project> python3 index_calculus.py
FAIL
FAIL
FAIL
FAIL
FAIL
FAIL
FAIL
FAIL
```

2.8 A Note on the Lower Bound of Generic DLP Algorithms

We have seen the algorithmic time complexities of the above algorithms, and also evidence of these time comparisons in our presentation. Now, we wish to continue the discussion from the first subsection, and establish a **lower bound** on the time complexity for the **Discrete Logarithm Problem.**We need to build on a few machinery first.

We define an **Encoding** of $(\mathbb{Z}_n, +)$ as an **injective map**

$$\sigma : \mathbb{Z}_n \to S$$

where S is a finite set. Here is an example:

Consider the group $(\mathbb{Z}_p^*, *)$ and let $\mathbb{Z}_p^* = \alpha >$. Let $n = p - 1$. We define the encoding σ as

$$\sigma(i) = \alpha^i \mod p \qquad 0 \leq i \leq n-1$$

We notice the following application of such an encoding:

Solving the DLP in $(\mathbb{Z}_p^*, *)$ with respect to $\alpha \equiv$ Solving DLP in $(\mathbb{Z}_n, +)$ with generator 1 under encoding σ

Now, our aim is to find a lower bound on the complexity of finding a discrete logarithm using any generic algorithm.

Suppose we have a random encoding σ for $(\mathbb{Z}_n, +)$. The DLP of a to the base 1 is a. Now, given σ we have the following with us:

- $\sigma(1)$ of the generator
- $\sigma(a)$ and
- a generic algorithm that computes the value of the discrete log a.

Here is a small aside:

$$\boxed{\text{Given } \sigma(i), \sigma(j), \text{ one can compute } \sigma((ci + dj) \mod n), c, d \in \mathbb{Z}_n}$$

We assume that we have access to a random oracle $\mathcal{O}$ that does all of this for us in $O(1)$ time. $\mathcal{O}$ computes the value of the discrete logarithm. We need to find a lower bound on its complexity. Consider a generic Discrete Logarithm algorithm below, in an experiment GENLOG.Here are the steps of the experiment:

1. The GENLOG sends across the encodings $\sigma_1 = \sigma(1), \sigma_2 = \sigma(2)$ to $\mathcal{O}$
2. $\mathcal{O}$ sends back a sequence of m encodings $\sigma_1, ...\sigma_m$ of linear combinations of 1 and a.
3. For each ordered pair (c_i, d_i), we have
$$\mathcal{O}(c_i, d_i) = \sigma_i = \sigma((c_i + d_i a) \mod n)$$
$$(c_1, d_1) = (1, 0)$$
$$(c_2, d_2) = (0, 1)$$
4. Since *sigma* is injective, we have
$$c_i + d_i a \equiv c_j + d_j a \mod n$$
$$\iff \sigma_i = \sigma_j$$
Since the ordered pairs are distinct, we have $d_j \neq d_j$ for $i \neq j$. Further, we assume n to be prime. Hence, we compute
$$a = (c_i - c - j)(d_i - d_j)^{-1} \mod n$$

Now, EXP-GENLOG is successful if and only if it outputs the correct value of a. Suppose GENLOG chooses a set
$$\mathcal{C} = \{(c_i, d_i) : 1 \leq m\} \subseteq \mathbb{Z}_n \times \mathbb{Z}_n$$
all at once before beginning its operation. Suppose it returns a set
$$GOOD(\mathcal{C}) = \{a \in \mathbb{Z}_n | a = (c_i - c - j)(d_i - d_j)^{-1} \mod n, i \neq j \in \{1, ..., m\}\}$$
We have
$$a \text{ can be computed } \iff a \in GOOD(\mathcal{C})$$
$$\implies |GOOD(\mathcal{C})| \leq \binom{m}{2}$$
$$\implies \mathbb{P}[a \in GOOD(\mathcal{C})] \leq \binom{m}{2}/n$$

Suppose GENLOG does not choose such a set, then it will compute a value of a randomly from the set $\mathbb{Z}_n - GOOD(\mathcal{C})$. Let $g = |GOOD(\mathcal{C})|$. We finally define the events

$$\begin{aligned} A &:= a \in GOOD(\mathcal{C}) \\ B &:= \text{ Algorithm returns correct value of } a. \end{aligned}$$

We find that

$$\begin{aligned} \mathbb{P}(B) &= \mathbb{P}(B|A)\mathbb{P}(A) + \mathbb{P}(B|\overline{A})\mathbb{P}(\overline{A}) \\ &= 1 \times \frac{g}{n} + \frac{1}{n-g} \times \frac{n-g}{n} \\ &= \frac{g+1}{n} \\ &\leq \frac{\binom{m}{2}+1}{n} \end{aligned}$$

Suppose GENLOG is a Las-Vegas type algorithm, and always returns the correct value of the discrete logarithm. Then, $P(B) = 1$, which implies that $m = \Omega(\sqrt{n})$. Therefore, we have learnt the following:

For a generic algorithm that computes the discrete logarithm correctly, it does so in the order $\Omega(\sqrt{n})$ where n is the order of the group.

3 Bibliography

1. Stinson, D. R., Paterson, K. G. (2019). Cryptography theory and practice. CRC Press.
2. Pollard, J. M. "Monte Carlo Methods for Index Computation $(\bmod\, p)$." Mathematics of Computation, vol. 32, no. 143, 1978, pp. 918–24. JSTOR, https://doi.org/10.2307/2006496. Accessed 2 May 2023.
3. Shi Bai and Richard P. Brent. 2008. On the efficiency of Pollard's rho method for discrete logarithms. In Proceedings of the fourteenth symposium on Computing: the Australasian theory - Volume 77 (CATS '08). Australian Computer Society, Inc., AUS, 125–131.